Computing the maximal canonical form for trees in polynomial time

Gunnar Brinkmann
Applied Mathematics, Computer Science and Statistics
Ghent University
Krijgslaan 281-S9,
9000 Ghent, Belgium
Gunnar.Brinkmann@UGent.be

January 19, 2018

Abstract

Known algorithms computing a canonical form for trees in linear time use specialized canonical forms for trees and no canonical forms defined for all graphs. For a graph $G = (V, E)$ the maximal canonical form is obtained by relabelling the vertices with $1, \ldots, |V|$ in a way that the binary number with $|V|^2$ bits that is the result of concatenating the rows of the adjacency matrix is maximal. This maximal canonical form is not only defined for all graphs but even plays a special role among the canonical forms for graphs due to some nesting properties allowing orderly algorithms. We give an $O(|V|^2)$ algorithm to compute the maximal canonical form of a tree.

keywords: tree, canonical form, structure enumeration

1 Introduction

A standard way to decide on the isomorphism of graphs is the use of *canonical representatives* of an isomorphism class. When canonical representatives are coded as numbers or strings, one can also interpret them as a *complete* graph invariant $i(G)$ – that is an invariant with the property that $i(G) = i(G')$ if and only if G and G' are isomorphic.

While for general graphs the complexity of the problem to determine a complete invariant is not yet known, for trees it is long known that the problem can be solved in linear time [7]. Complete invariants for trees that can be computed in linear time are special invariants defined only for trees and not invariants defined for all graphs where the restriction to the set of trees can be computed in linear time.

Even among the canonical forms defined for all graphs, some canonical forms – e.g. the *maximal form* – play a special role. The maximal form of a graph is the isomorphic graph G' with vertices $1, \ldots, |V|$ for which the binary number $b(G')$ with $|V|^2$ bits obtained by concatenating the rows of the adjacency matrix is maximal. Equivalently one can consider the number $b(G')$ or the string of zeros and ones coding the graph as the maximal form. Instead of concatenating all rows to form a binary number, one can also restrict oneself to the $(n^2-n)/2$ bits of the upper right triangle without the diagonal (that is: for $1 \leq i \leq n$ use only the last $n-i$ bits of row i) and require maximality of the binary number obtained that way.

It is not only that this canonical form is very intuitive and easy to explain, it also has special *nesting* properties necessary for orderly algorithms: If $G = (V, E)$ is a graph in maximal form and e the edge corresponding to the last bit in the binary number obtained from the upper triangle of the adjacency matrix, then $G' = (V, E - e)$ is also in maximal form. This property allows *orderly algorithms* for the generation of complete and isomorph free lists of graphs, see e.g. [1], [3], [4], [6], [8].

From the theoretical point of view a polynomial algorithm for this canonical form is interesting in mathematics and algorithmics. From the practical point of view it is especially interesting in mathematical chemistry: as the graphs studied in mathematics are mostly either trees or graphs with many cycles (e.g. regular graphs with degree at least 3), also structure enumeration is especially interesting for these classes. For trees specialized generators exist [5]. Generators for graphs using the maximal form work recursively, so the importance of the algorithm to compute the maximal form for a tree depends on the place in the recursion where trees occur. Though some programs like e.g. [4] allow the generation of graphs with few cycles, the focus lies on graphs with many cycles and in that case trees occur at an early stage of the recursion, so that the complexity of computing the canonical form of the trees has not much impact on the running time. In chemistry graphs with few cycles and even monocyclic graphs have been well studied and even generators for subclasses of monocyclic graphs have been developed (see e.g. [2], [9] for just two of many examples). When generating these classes, the trees can occur until late in the recursion and can have an important impact on the running time, so that a polynomial test can also be of practical importance.

This maximal form can also be defined in other, equivalent ways: One can define a string by listing the neighbours of the vertices $1, \ldots, n$ (in this order) in increasing order, separated by a symbol that is considered to be larger than any vertex number. Then a graph is in maximal form if and only if this string is lexicographically minimal. Similar to using only the upper right triangle of the adjacency matrix, one can also restrict the string and list for each vertex v only the neighbours $w > v$. We call this way of representing the labelled graph the *reduced string representation*.

2 The algorithm for trees

The vertex labels in a maximal form of a graph G can be obtained by a *breadth first* numbering starting with the vertex labelled 1. This remains true if one does not look for the graph with largest $b(G')$ among all isomorphic graphs, but only among those where a certain vertex of G can be mapped onto the vertex labelled 1.

For trees with n vertices a breadth first labelling implies that each label $2, \ldots, n$ occurs exactly once in the reduced string representation and that the labels occur in increasing order. So the reduced string representation of a tree is completely described by the positions of the separation symbols and we can as well describe the string as a 0-1 sequence with 0 representing a label and 1 representing a separator. A reduced string representation is minimal if and only if the corresponding 0-1 sequence is also minimal. We will call 0-1 sequences corresponding to reduced string representations 0-1-representations.

Let z_i, resp. o_i denote the number of zeros, resp. ones on positions 0 to i in a 0-1-representation. As each 0 in the 0-1 sequence stands for a vertex whose list of neighbours ended with a 1 follows later, we have $z_i \geq o_i$ for each position i except the last one and $z_i < o_i$ if i is the last position. This implies that no 0-1-representation is a proper starting sequence of another 0-1-representation – they are either identical or differ in a position that is present in both representations. The vertices at a given distance d of the vertex labelled 1 get consecutive numbers and we call these vertices and the corresponding entries and separators after these entries of the 0-1-representation the d-th *level*.

We will also work with 0-1-1′-representations, which are strings with symbols 0, 1 and 1′. The 0-1-1′-representation of a tree is equal to its 0-1-representations except for the fact that each last 1 on a level is replaced by a 1′ which is considered equal to a 1 when comparing strings in order to find the lexicographically smallest one. These representations are only used to simplify the description. As the 1's at the end of a level can easily be computed from the 0-1-representation, there is a simple one-to-one correspondence between these representations and even the algorithm could be rewritten to work as efficiently on 0-1-representations – but would look more complicated. All results and remarks valid for 0-1-representations are also valid for 0-1-1′-representations.

These 0-1-representations or 0-1-1′-representations are no canonical forms defined for all graphs, but they are just intermediate steps in the algorithm from which the general canonical form can be easily reconstructed (e.g. in time $O(|V|)$ for the reduced string representation or in time $O(|V|^2)$ when the adjacency matrix representation is asked).

Each 0-1-representation corresponds to one or more labellings of the tree from which it can be obtained. Together with a 0-1-representation we

always assume a corresponding labelling to be given. In rooted trees we will only consider labellings that assign number 1 to the root and only consider representations that come from such labellings.

We interpret each edge $\{u, v\}$ of a tree as two directed edges (u, v) and (v, u) and denote the component of $T - \{u, v\}$ containing v and rooted at v as $T(u, v)$.

Lemma 2.1. *If $s(T_v)$ is the minimal* 0-1*-representation of a tree T_v rooted at v and u a neighbour of v, then labelling the vertices of $T(v, u)$ in the same order as for the minimal* 0-1*-representation of T_v gives a minimal* 0-1*-representation of $T(v, u)$.*

Proof. Assume that the minimal 0-1-representation $s(T(v, u))$ is smaller than the representation $\bar{s}(T(v, u))$ induced by $s(T_v)$. Then there is a position k so that positions 1 to $k - 1$ of $s(T(v, u))$ and $\bar{s}(T(v, u))$ agree, position k of $s(T(v, u))$ is 0 and position k of $\bar{s}(T(v, u))$ is 1.

Representation $s(T(v, u))$ also corresponds to a breadth first labelling, so the entries of the 0-1-representation corresponding to vertices at a fixed distance $d > 0$ of u form a substring of $s(T(v, u))$ as well as of $s(T_v)$. The length of the substring is given by the sum of all degrees of vertices at distance $d-1$ of u, so it is independent of the specific breadth first numbering inside $T(v, u)$. If we keep the labels of all vertices of T_v that are not in $T(v, u)$ and take another breadth first order in $T(v, u)$, the set of positions in $s(T_v)$ corresponding to vertices of $T(v, u)$ remains the same. If we replace the order in which the vertices inside $T(v, u)$ are labelled by the order in which they are labelled for $s(T(v, u))$, the resulting string $s_m(T_v)$ is identical with $s(T(v, u))$ until the k-th position corresponding to an element of $T(v, u)$. This position is 1 in $s(T_v)$ and is 0 in $s_m(T_v)$ – so $s(T_v)$ was not minimal – a contradiction.

□

For a set $\{s_1, \ldots, s_k\}$ of 0-1-1′-representations Algorithm 1 defines how the result $m(s_1, \ldots, s_k)$ of merging the strings is obtained. Before starting the merging process, $\{s_1, \ldots, s_k\}$ are lexicographically sorted and put in a list S in increasing order. For each string s we have an index p_s giving the next position in s to be read (initially 0) and a variable l_s giving the number of symbols in s. We will denote the symbol at position i of a string s as $s[i]$.

If $s_i < s_j$ and both contain a level k, then the entries of s_i on level k are filled in before the ones of s_j. Applying this merging to strings coming from labelling trees guarantees that the labelling comes from a breadth first numbering.

With $L = \sum_{i=1}^{k} |s_i|$ the strings $s_1, \ldots, s_k$ can be sorted in time $O(L)$ by using e.g. radix sort and taking the lengths of the sequences into account, so that sequences are only considered for indices that are present in the list.

Algorithm 1 An algorithm to merge 0-1-1′-representations.

1: $m(s_1, \ldots, s_k) = \underbrace{0, 0, \ldots, 0}_{k \text{ times}}, 1'$
2: **while** S is not empty **do**
3: **for** all $s \in L$ in increasing order **do**
4: **while** $s[p_s] \neq 1'$ **do**
5: append $s[p_s]$ to $m(s_1, \ldots, s_k)$
6: set $p_s = p_s + 1$
7: **end while**
8: append 1 to $m(s_1, \ldots, s_k)$
9: set $p_s = p_s + 1$
10: **if** $p_s \geq l_s$ **then**
11: remove s from the list
12: **end if**
13: **end for**
14: replace the last 1 of $m(s_1, \ldots, s_k)$ by $1'$
15: **end while**

As also the construction of $m(s_1, \ldots, s_k)$ from the sorted strings can be done in time $O(L)$, the whole computation of $m(s_1, \ldots, s_k)$ takes time $O(L)$.

Lemma 2.2. *If $s_1, \ldots, s_k$ are minimal* 0-1-1′*-representations of trees rooted at $v_1, \ldots, v_k$, and T_v is the tree rooted at v obtained from these trees by adding the new vertex v and connecting it to $v_1, \ldots, v_k$, then $m(s_1, \ldots, s_k)$ is the minimal* 0-1-1′*-representation $s(T_v)$ of T_v.*

Proof. By Lemma 2.1 we know that also $s(T_v)$ induces minimal representations of $T(v, v_1), \ldots, T(v, v_k)$, so these representations are identical to $s_1, \ldots, s_k$.

Assume that the order in which $v_1, \ldots, v_k$ are labelled $2, \ldots, k+1$ for $s(T_v)$ is $v_{p_1}, \ldots, v_{p_k}$. What remains to be shown is that this order is – up to interchanging vertices v_i, v_j with $s_i = s_j$ – identical to the order used by the algorithm to compute $m(s_1, \ldots, s_k)$. This means that we have to show that for $1 \leq i < k$ we have that $s_{p_i} \leq s_{p_{i+1}}$.

Assume that there is an i so that $s_{p_i} > s_{p_{i+1}}$ and that the first level on which they differ is level l_d. For each level l of $T(v, v_{p_i})$ and $T(v, v_{p_{i+1}})$, the substrings follow each other on level $l+1$ of $s(T_v)$. If we keep the order in which the vertices are labelled inside $T(v, v_1), \ldots, T(v, v_k)$, but label $v_{p_{i+1}}$ with i and v_{p_i} with $i+1$, then the new representation $s'(T_v)$ differs from $s'(T_v)$ for the first time on level $l_d + 1$ – and here a string $s_1 s_2$ with $s_2 < s_1$ is replaced by $s_2 s_1$ which is lexicographically smaller. So $s'(T_v) < s(T_v)$ contradicting the minimality of $s(T_v)$.

□

In order to compute the minimal representation $s(T(u,v))$ of all directed edges (u,v) in a tree, we still need some preprocessing:

If $d(u,v)$ denotes the maximum distance of v from a leaf of T in $T(u,v)$, then for all directed edges (v,w) with $w \neq u$ we have $d(v,w) < d(u,v)$. This implies that once we have computed $s(T(e))$ for all directed edges e with $d(e) \leq d-1$, we can compute $s(T(e))$ for all directed edges e with $d(e) \leq d$.

A straightforward breadth first algorithm starting at each edge could compute $d(u,v)$ for all edges in together $O(|V|^2)$ steps – which is inside the time limits of the whole algorithm – but also a speedup in parts can be useful and interesting. In Algorithm 2 we give a linear time algorithm.

First each vertex v is assigned a counter $c(v)$ initialized as $\deg(v)$ counting the outgoing edges which have not yet been processed. All edges are marked as not yet been processed and all edges (u,v) with $\deg(v) = 1$ are added to a list D and assigned value $d(u,v) = 0$. Furthermore each vertex v has a list $S(v)$ of directed edges starting at v and a list $E(v)$ of directed edges ending in v.

Algorithm 2 An algorithm to compute $d(u,v)$.

```
 1: while D is not empty do
 2:     let (u,v) be the first element of D
 3:     remove (u,v) from the list and mark (u,v) as processed
 4:     set c(u) = c(u) − 1
 5:     if c(u) = 1 then
 6:         let (w,u) ∈ E(u) be the edge so that (u,w) is not yet processed
 7:         set d(w,u) = d(u,v) + 1
 8:         append (w,u) to the end of D
 9:     else if c(u) = 0 then
10:         for all (w,u) ∈ E(u), w ≠ v do
11:             set d(w,u) = d(u,v) + 1
12:             append (w,u) to the end of D
13:         end for
14:     end if
15: end while
```

Lemma 2.3. *Algorithm 2 needs $O(|V|)$ steps and computes $d(u,v)$ for all directed edges of an input tree $T = (V,E)$.*

Proof. The fact that Algorithm 2 is linear in $|V|$ is immediate, except for lines 6 and lines 10 to 12. Line 6 is executed once for each vertex u and takes time $\Theta(\deg(u))$. The same is true for the loop in lines 10 to 12. So altogether also these lines take time $\Theta(\sum_{v \in V} \deg(v))$ and $\sum_{v \in V} \deg(v) = 2|V| - 2$ for trees.

In order to prove that Algorithm 2 computes $d()$ correctly we first observe that the value of $d()$ grows monotonically with the time that directed edges

are added to D. So when $c(u) = 1$, then only one edge (u, w) incident with u has not yet been processed and for the inverse edge (w, u) the value of $d()$ is one larger than the largest value of the edges that have been processed – which is the value of the last one processed. An analogous argument holds for the case when $c(u) = 0$.

So all values of $d()$ that are filled in are correct and it only remains to show that for all directed edges e the value of $d(e)$ is filled in:

In the beginning the value $d(e) = 0$ is filled in for all edges e with $d(e) = 0$ and the edges are added to D, so let us assume that for all edges e with $d(e) < m$ the value is filled in and that they are added to D at some time. Let (u, v) be an edge with $d(u, v) = m$. Then all edges (v, w) starting at v except maybe (v, u) have $d(v, w) < m$, so they are added to the list at some time. When the last of these edges is processed, we have $c(v) \leq 1$ and the value of $d(u, v)$ is filled in, proving by induction that for all edges e the value of $d(e)$ is filled in.

□

As $d(u, v) \leq |V| - 2$ for all (u, v), we can sort the directed edges in increasing order of the values of $d()$ in time $O(|V|)$ by using bucket sort. This way we produce a sorted list of directed edges $(u_1, v_1), \ldots, (u_{2|V|-2}, w_{2|V|-2})$.

Algorithm 3 An algorithm to compute $s(T(u, v))$.

```
1: set i = 1
2: while d(u_i, v_i) = 0 do
3:     s(T(u_i, v_i)) = 1'
4:     set i = i + 1
5: end while
6: while i <= 2|V| - 2 do
7:     s(T(u_i, v_i)) = m(s(T(v_i, w_1)), ..., s(T(v_i, w_k)))
8:     with (v_i, w_1), ..., (v_i, w_k) the directed edges starting at v_i that are
   not (v_i, u_i)
9:     set i = i + 1
10: end while
```

There are $2|V| - 2$ directed edges and in each merging step the sum of the lengths of the lists is at most $2|V| - 1$, so all merging steps together can be done in time $O(|V|^2)$.

Finally we can define $s(v)$ for all $v \in V$ as $s(v) = m(s(T(v, w_1)), \ldots, s(T(v, w_k)))$ with $(v, w_1), \ldots, (v, w_k)$ all edges starting at v.

We have $|V|$ merging operations with each a cost of $O(|V|)$, so again a total cost of $O(|V|^2)$. By Lemma 2.2 these $s(v)$ (with $1'$ replaced by 1) are the minimal 0-1-representations of the tree rooted at v. The smallest string among all $s(v)$ is the minimal 0-1-representation. It can obviously be found in time $O(|V|^2)$. Storing the information on the order of the sorted strings

in the various sorting steps would allow to reconstruct the labelling that leads to the minimal representation in linear time.

As a consequence of the lemmas and algorithms we get

Theorem 2.4. *The maximal canonical form of a tree $T = (V, E)$ can be computed in time $O(|V|^2)$.*

3 Conclusions and possible further work

The algorithm does not have the same linear time bound as algorithms for specialized canonical forms for trees, nevertheless the algorithm is polynomial of a small order and fast for practical applications. It would even be optimal up to a constant factor if the input would be the adjacency matrix of the graph or the required output would be the maximal adjacency matrix. It would be very interesting to know whether an algorithm that takes time $o(|V|^2)$ to compute the canonically labelled tree or the 0-1-representation exists.

For application in orderly algorithms it would also be interesting to know whether there are still polynomial time algorithms to compute the maximal canonical form if a small, bounded number of cycles is allowed. It is easy to design such algorithms for canonical forms designed extra for this application. For the maximal canonical form it does not even look simple for monocyclic graphs.

It would also be interesting to know whether for other general canonical forms polynomial algorithms for trees exist, e.g. for the canonical form requiring minimality of the string obtained from the lower left triangle of the adjacency matrix. This canonical form is proven to be NP-complete for general graphs, but for trees a polynomial algorithm could be possible.

References

[1] G. Brinkmann. Fast generation of cubic graphs. *Journal of Graph Theory*, 23(2):139–149, 1996.

[2] Z. Du. Wiener indices of trees and monocyclic graphs with given bipartition. *International Journal of Quantum Chemistry*, 112:1598–1605, 2012.

[3] I.A. Faradžev. Constructive enumeration of combinatorial objects. *Colloques internationaux C.N.R.S. No260 - Problèmes Combinatoires et Théorie des Graphes, Orsay 1976*, pages 131–135, 1976.

[4] R. Grund. Konstruktion schlichter Graphen mit gegebener Gradpartition. *Bayreuther Mathematische Schriften*, 44:73–104, 1993.

[5] G. Li and F. Ruskey. The advantages of forward thinking in generating rooted and free trees. pages 939–940, 1999. 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).

[6] M. Meringer. Fast generation of regular graphs and construction of cages. *Journal of Graph Theory*, 30(2):137–146, 1999.

[7] R.C. Read, editor. *Graph Theory and Computing*. Academic Press, New York, 1972.

[8] R.C. Read. Every one a winner. *Annals of Discrete Mathematics 2*, pages 107–120, 1978.

[9] M. Suzuki, H. Nagamochi, and Akutsu T. Efficient enumeration of monocyclic chemical graphs with given path frequencies. *Journal of Cheminformatics*, 6(31), 2014.